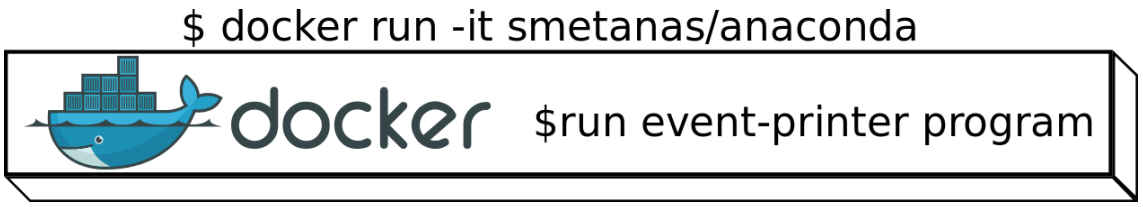
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2016
http://excel.fit.vutbr.cz

ANaConDA v kontejneru nebolí

Štěpán Smetana*



Abstrakt
Testování paralelních programů je problematické, protože paralelní chyby se nemusí projevit při každém běhu programu. Jednou z úspěšných technik ulehčující testování paralelních programů je metoda vkládání šumu. Tato technika je implementována např. ve frameworku ANaConDA (C/C++ na binární úrovni) nebo Race Detector & Healer (Java). Bohužel Instalace frameworku ANaConDA je pro jeho typického uživatele komplikovaná. Článek řeší tento problém pomocí technologie linuxových kontejnerů v prostředí Docker, a tak lze pomocí dvou jednoduchých příkazů spustit dynamickou analýzu. Dále se článek věnuje rozšíření frameworku v podobě nového analyzátoru.

Klíčová slova: ANaConDA — Dynamic analysis — Docker — Eraser — Intel PIN

Přiložené materiály: Kontejner na Docker hub - smetanas/anaconda

*xsmeta04@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Úvod

Paralelní programy obsahují chyby, které není jednoduché správně identifikovat. Souběhové chyby se nemusí vyskytnout při každém běhu programu, a přestože se vyskytnou, tak program většinou nemůže poznat, že pracuje s nesprávnými daty. Analyzátor proto musí sledovat jednotlivé instrukce a zpravidla být svědkem souběhové chyby, aby mohl jednoznačně určit, že muže dojít k takové chybě. Existují techniky, jak zvýšit pravděpodobnost odhalení chyb. Jednou z nich je metoda vkládání šumu, která je implementována např. ve frameworku ANaConDA (analýza C/C++ programů na binární úrovni) nebo Race Detector & Healer (Java).

Instalace frameworku ANaConDA je komplikovaná pro cílového uživatele. Aby bylo jednodušší používat framework ANaConDA, je snaha tento software zapouzdřit.

Framework ANaConDA obsahuje několik základních analyzátorů, jmenovitě např. AtomRace, Goodlock. Je potřeba tuto sadu rozšiřovat, aby se pokrylo co nejvíce odlišných chyb a zabránilo falešným hlášením.

Hlavním problémem instalace prostředí ANaConDA jsou jeho závislosti (např. GCC, libelf, libboost). I když se instalátor snaží většině problémům předcházet, tak v některých případech instalace nemusí být zcela triviální záležitostí a stává se, že bez dalších vnějších zásahů by instalaci nebylo možné dokončit. Většinou jsou tyto problémy zaviněny použitím nějakého novějšího/zastaralého softwaru, nebo konfliktem s již nainstalovaným softwarem, případně nedodržením nějakého standardního formátu, se kterým instalátor počítal.

Řešením je zkompletovat celé prostředí ANaConDA do jednoho uceleného balíku, který bude obsahovat všechny závislosti, a nebude tak závislý na jiném nainstalovaném softwaru mimo tento balík. To je možné realizovat například jako Docker kontejner.

Výsledkem této práce je zapouzdření frameworku ANaConDA do jednotného balíku, který spustí cílový uživatel pomocí jednoho příkazu. Navíc byl implementován další analyzátor Eraser.

2. Docker kontejnery

Virtualizace kontenerů je ve světě počítačů už skoro 20 let. Práce s nimi však nebyla jednoduchá a byla časově náročná. To se změnilo s příchodem Dockeru. Ten umožňuje uživateli s minimální znalostí jednoduše nasadit kontejner ve formě obrazu pro Docker. Nejdůležitější částí je možnost lehce vytvářet, spravovat a sdílet vlastní obrazy pro Docker. Veřejným uložištěm pro tyto obrazy je Docker hub.

2.1 Komponenty Dockeru

Docker se skládá z několika hlavních částí:

- Docker engine - klient-serverová aplikace umožňující virtualizaci kontejnerů.
- Obrazy pro Docker - šablony, složeny z vrstev, pomocí kterých jsou vytvářeny Docker kontejnery.
- Docker kontejner - vrstva vytvořená na vrcholu bázového obrazu pro Docker. Mezi základní operace s kontejnery patří spouštění, zastavování, mazání.
- Docker registr - služba umožňující hosting obrazů pro Docker. Jsou dva základní typy registrů: veřejné a soukromé.

2.2 Jmenné prostory, cgroups a chroot

Docker používá linuxové jmenné prostory, aby zajistil dostatečnou bezpečnost. Vlastní jmenný prostor je vytvořen pro každý běžící kontejner. Linuxové jádro poskytuje funkce `cgroups` [1], které umožňují omezit a stanovit priority zdrojů (CPU, paměť, síťě, atd.). Tyto funkce umožňují robustní izolaci aplikace, např. vlastní skupiny PID, hostname, userid, síť, souborový systém. Kontejnery poskytují každé aplikaci nezávislé běhové prostředí, přičemž se vyhne drahé režii plnohodnotného virtuálního stroje.

2.3 Virtuální stroj vs. kontejner Docker

Kontejnery mají podobné výhody v izolaci a přidělování zdrojů jako virtuální stroje, ale odlišný základní přístup umožňuje kontejnerům být mnohem lépe přenosnými a efektivnějšími. Na rozdíl od hypervirtualizace, kdy jeden a více virtuálních strojů běží nezávisle na jednom fyzickém hardwaru přes zprostředkovanou vrstvu, kontejnery namísto toho běží v uživatelském prostoru na vrcholu jádra operačního systému (viz obrázek 1). Díky tomu jsou však někdy kontejnery považovány za méně flexibilní, protože obecně lze spustit pouze kontejner založený na stejném nebo podobném operačním systému jako hostitelský.

3. Framework ANaConDA

Framework ANaConDA [2] (**A**daptable **Na**tive-code **Con**currency-focused **D**ynamic **A**nalysis) je prostředí, které slouží jako podpora při programování dynamických analyzátorů pro analýzu vícevláknových C/C++ programů na binární úrovni. Framework přebírá některé problémy, a tak se návrháři a programátoři mohou soustředit pouze na tvorbu nových analyzátorů. Framework mimo jiné obsahuje také základní sadu analyzátorů.

3.1 Vlastnosti

ANaConDA má možnost při běhu testované aplikace kontrolovat některé události, jako například:

- přístupy do paměti,
- synchronizace vláken,
- spuštění nebo ukončení vláken,
- zpracování vyjímek nebo
- posloupnosti volání.

3.2 Architektura prostředí ANaConDA

ANaConDA[1] je postavena na prostředí PIN (Intel)[2] pro instrumentaci binárního kódu. ANaConDA tak plně podporuje pouze binární soubory kompatibilní s instrukcemi procesorů Intel. ANaConDA se skládá z několika částí (jak je zobrazeno na obrázku 3):

- prostředí PIN potřebné pro připojení se ke sledovanému programu v rámci testu,
- jeden či více analyzátorů,
- jádro ANaConDA, které ovládá připojení a zpětné volání analyzátoru.

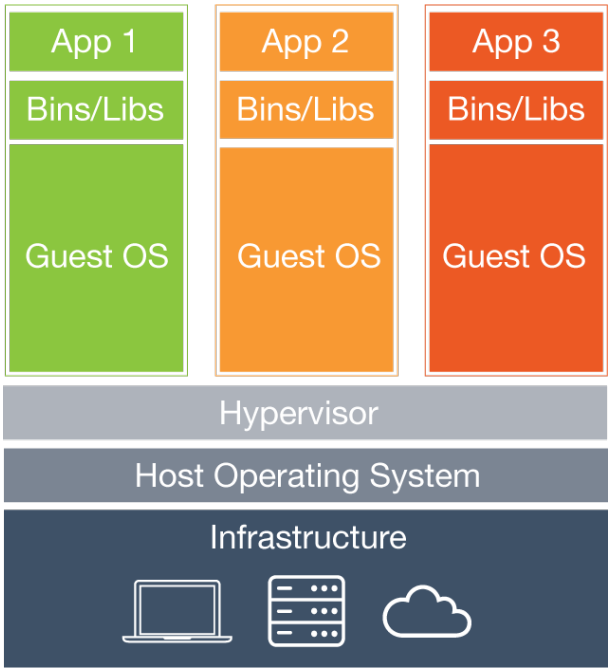
Jádro ANaConDA navíc používá knihovnu libDIE na extrakci ladících informací, která využívá knihovny libdwarf [3] a libelf [4].

3.2.1 Intel PIN

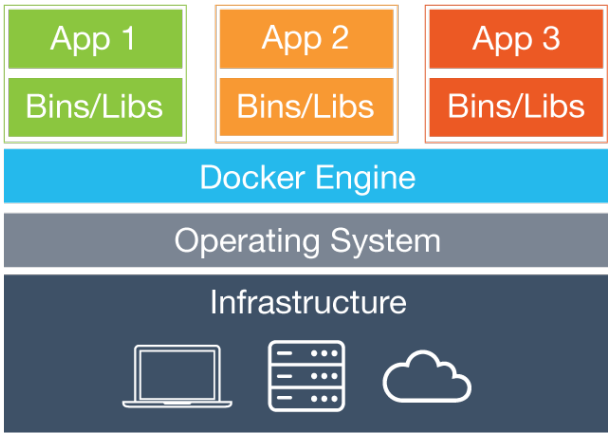
Intel PIN je nástroj podporující dynamickou instrumentaci binárního programu. Prostředí PIN používá JIT (Just-In-Time) přístup, který vytvoří a dynamicky upraví kopii aplikace. U tohoto přístupu není prováděn původní kód. Díky tomu je schopen se PIN připojit i na už běžící program a zvládne zpracovat dynamicky generovaný kód. PIN používá i tzv. sondovací režim (statická binární instrumentace), kdy upraví původní instrukce v programu a vloží skoky do kódu programu, což má nižší režii [5].

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/.en

[2] https://software.intel.com/en-us/articles/pintool



Obrázek 1. Virtuální stroj obsahuje aplikace, potřebné binární soubory a knihovny a celý hostující operační systém, což mohou být i desítky GB.



Obrázek 2. Kontejnery obsahují aplikace a všechny její závislosti, ale sdílejí jádro s ostatními kontejnery. Běží jako izolované kontejnery v uživatelském prostoru na hostitelském operačním systému. Taky nejsou vázány na žádnou konkrétní infrastrukturu - Docker kontejnery běží na kterémkoliv počítači, jakékoli infrastruktuře a v každém cloudu.

3.3 Analyzátory

Framework ANaConDA obsahuje několik základních analyzátorů:

- event-printer - výpis všech událostí,
- atomrace [6] - detekce datarace (algoritmus vyvinutý skupinou VeriFIT[3]),
- eraser [7] (popsáno v kapitole 4),
- goodlock [8] - detekce uváznutí,
- contract-validator - aktuální výzkum VeriFITu[3],
- hldr-detector a tx-monitor.

3.4 Metoda vkládání šumu

Vkládání šumu se pokouší zvýšit šanci být svědkem ojedinělého běhu vedoucímu k chybě tím, že ovlivňuje přepínání vláken programu. Tato metoda může vkládat šum náhodně, nebo dodržovat předem stanovenou heuristiku. Framework ANaConDA umožňuje ovlivňovat přepínání vláken pomocí `yield()` nebo `sleep()`. Dále je možné upřesnit, při jakých událostech se má vkládaný šum generovat. ANaConDA umožňuje vložit šum s různým nastavením k přístupům k paměti nebo k jakýmkoli sledovaným operacím.

4. Rozšíření o analyzátor Eraser

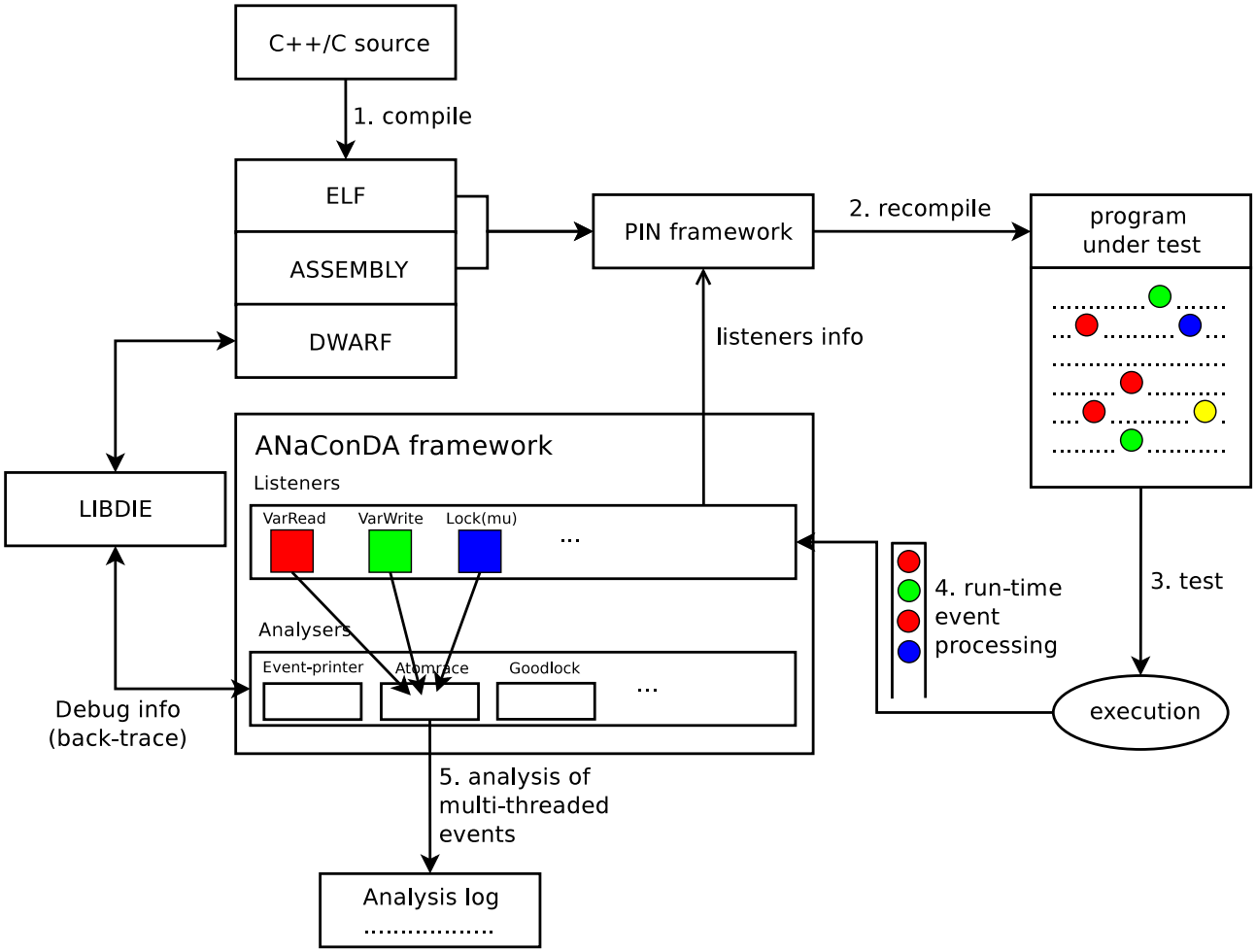
Algoritmus eraser [7] je založen na správném dodržování zámkové disciplíny. Patří mezi algoritmy, u kterých se počítá s tím, že budou hlásit falešné chyby. Hlásí chybu, pokud neexistuje zámek, který by byl u každého přístupu na konkrétní místo ve sdílené paměti. Pokud je přistupováno k paměti více vlákny a alespoň jeden přístup byl zápis, tak se jedná o sdílenou paměť. Pokud k této části paměti bylo přistupováno k zápisu více než jedním vláknem, tak může dojít k chybě souběhu, jestliže při každém přístupu do této paměti není držen vzájemně vylučující se zámek.

4.1 Implementace

Prvním bodem imlementace je registrace funkcí, jak je vidět na obrázku 4. Volání těchto funkcí bude vloženo do programu při znovupřeložení pomocí frameworku PIN. V případě analyzátoru Eraser je potřeba sledovat operace nad pamětí a zámky. Protože atomický přístup nemůže způsobit časově závislou chybu nad daty (datarace), sledujeme pouze neatomické čtecí a zápisové přístupy. Algoritmus, podle kterého je implemetován tento analyzátor, je založen na dodržování zámkové disciplíny, a tak je třeba sledovat operace se zámky (zamčení a odemčení).

Druhým bodem je samotná implementace algoritmu. Každému vláknu je přiřazena množina aktuálně aktivních zámků. Každé proměnné (paměťové lokaci) je přiřazena množina držených zámků. Při každém zamknutí zámku se přiřadí daný zámek k vláknu, které je aktuálně aktivní. Při odemknutí zámku se daný zámek odstraní z aktuálně aktivního vlákna. Při přístupu k proměnné je nejprve aktualizována množina zámků, které byly u každého výskytu této proměnné

[3] http://www.fit.vutbr.cz/research/groups/verifit/



Obrázek 3. Schéma frameworku ANaConDA. Framework ANaConDA nemusí mít přístup ke zdrojovým souborům. Je však vhodné mít zkompilovaný program s ladícími informacemi, aby analýza poskytnula co nejvíce informací. Po spuštění analýzy PIN framework provede znovupřeložení programu s tím, že podle potřeb analyzátoru vloží do programu vlastní instrukce. Nyní dojde ke spuštění programu. Instrukce vložené pomocí PINu pak vždy za běhu programu spustí určitou obsluhu, která je zpracována analyzátorem. Analyzátor má přístup k základním informacím z formátu ELF nebo registrů. Pokud analyzátor potřebuje získat více informací, tak ANaConDA podporuje stopu volání ze zásobníku (back-trace) funkcionálně shodnou s Linuxovou funkcí `backtrace()`, která obsahuje návratové adresy právě aktivních funkcí. Dále může analyzátor získat ladící informace pomocí knihovny libDIE. Výsledkem analýzy je záznam popisující nastavení frameworku a událostí zaznamenaných analyzátorem.

(průnik s množinou zámků držených aktivním vláknem). Dále je aktualizován stav dané proměnné (viz. obrázek 5). Pokud má paměť prázdnou množinu stavů a je ve stavu $Shared-Modified$, tak může vzniknout datarace nad touto proměnnou. Proměnná může přejít do stavu *Reported*, abychom nevypisovali už ohlášenou chybu.

5. ANaConDA jako kontejner pro Docker

Docker je ideálním prostředí na zapouzdření, provoz a správu aplikací flexibilním a efektivním způsobem. Jeho přístup k virtualizaci kontejnerů se rozšiřuje mezi OS a poskytovateli cloud computingu jako je RedHat, Microsoft a Amazon.

Aplikace distribuovaná jako obraz pro Docker zahrnuje všechny závislosti a konfigurace potřebné k běhu, čímž koncový uživatel nemusí instalovat balíčky a řešit problémy se závislostmi. Docker poskytuje nástroje potřebné k sestavení, provozu a správě aplikace zabalené jako obraz pro Docker. Jsou dva způsoby sestavení:

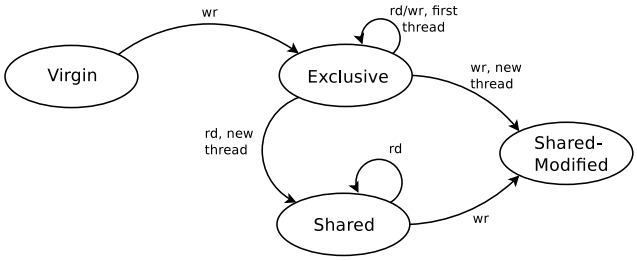
- spustit v Docker nějaký základní obraz operačního systému, nainstalovat všechny potřebné nástroje a uložit kontejner,
- vytvořit `Dockerfile` - dokumentovat kroky potřebné k sestavení obrazu.

```
PLUGIN_INIT_FUNCTION()
{
    // Register callback functions called before access events
    ACCESS_BeforeMemoryRead(beforeMemoryRead);
    ACCESS_BeforeMemoryWrite(beforeMemoryWrite);

    // Register callback functions called before synchronisation events
    SYNC_BeforeLockAcquire(beforeLockAcquire);
    SYNC_BeforeLockRelease(beforeLockRelease);

    // Initialise R/W mutex for guarding access to global variables
    PIN_MutexInit(&g_MapMutex);
}
```

Obrázek 4. Funkce, která registruje obsluhy operací před čtením a zápisem do paměti. Dále registruje obsluhy operací před zamčením a odemčením zámku. Nakonec registruje sdílený zámek na ochranu globálních proměnných. Tyto obsluhy jsou volány při každém výskytu dané události.



Obrázek 5. Eraser udržuje stavy všech proměnných, ke kterým bylo přistoupeno. Nově alokovaná paměť má stav *Virgin*. Jak různé vlákna přistupují k proměnné, tak se její stav mění podle závislostí zobrazených v obrázku. Souběhové chyba může nastat u proměnnou ve stavu $Shared-Modified$. Převzato z [7].

5.1 Obraz pro Docker pomocí Dockerfile

Dockerfile je soubor popisující pravidla pro automatický způsob vytvoření obrazu pro Docker. Základní příkazy pro tvorbu obrazů pro Docker pomocí Dockerfile jsou následující (zobrazeny v rámečcích):

```
FROM ubuntu:vivid
```

Instrukce určující Docker, aby využil veřejný obraz Ubuntu jako základ pro nový obraz. Díky tomu nemusíme vytvářet vlastní obrazy OS. Dále zde specifikujeme verzi Ubuntu 15.04 Vivid. Tuto verzi používáme proto, že její základní balík obsahuje minimální požadovanou verzi GCC na správný běh frameworku ANaConDA a zároveň je dostatečně stabilní.

```
ADD ./anaconda/ /root/anaconda/
```

Tato instrukce zkopíruje soubory frameworku ANaConDA dovnitř obrazu.

```
RUN apt-get update && apt-get install -y \
build-essential ...
```

Nyní jsou nainstalovány všechny potřebné balíky, na kterých je framework závislý. Pak je nainstalována samotný framework a nakonec vymazány pomocné subory.

```
ENTRYPOINT install
```

Nakonec je určen přístupový bod. Ten spustí skript, který oznámí koncovému uživateli, že bude používat software třetí strany PIN a odkaz na licenci tohoto softwaru. Pokud s tím uživatel souhlasí, tak je spuštěna instalace PINu.

5.2 Použití

V případě, že uživatel nemá Docker, je potřeba jej nainstalovat. Jsou dvě možnosti, jak povolit, aby PIN mohl pak upravovat běžící procesy uvnitř Dockeru. První možností je změnit profil AppArmoru[4] pro Docker (dočasně nebo natrvalo). Druhou možností je povolit to pomocí vstupního parametru `--security-opt`. Paramerem `-v ./:/root/` určíme, že aktuální adresář bude sdílený s kontejnerem a bude přístupný ve složce `/root`. Nyní můžeme spustit Docker kontejner pomocí příkazu:

```
docker run –security-opt apparmor:unconfined
-v ./:/root/ -it smetanas/anaconda
```

Po potvrzení instalace softwaru PIN můžeme spustit analýzu:

```
run event-printer ‘find /path/to/static/directory’
```

Výslednou analýzu lze získat z příkazového řádku, uložením do sdílené složky, výpisem na port, nebo pomocí příkazu:

```
docker cp [OPTIONS] CONTAINER:SRC_PATH
DEST_PATH
```

[4] http://wiki.apparmor.net/index.php/Main_Page

6. Závěr

Příchod vícejádrových procesorů a vícejádrové programování se stalo běžně používané ve všech jazycích. To však otevřelo prostor pro chyby způsobené špatnou synchronizací vláken. V tomto článku byl popsán framework ANaConDA, který je určen na dynamickou analýzu vícevláknových programů v C/C++.

Dále byl popsán analyzátor Eraser a v neposlední řadě vytvoření jednotného balíku pro prostředí Docker. Tento balík umožňuje velice snadno pomocí dvou příkazů spustit analýzu programu bez nutnost složité instalace.

```
docker run –security-opt apparmor:unconfined
-v ./:/root/ -it smetanas/anaconda
```

```
run event-printer ‘find /path/to/static/directory’
```

Díky této práci si může kdokoli stáhnout obraz pro Docker z veřejného repozitáře umístěném na Docker Hub (smetanas/anaconda).

Poděkování

Chtěl bych poděkovat Ing. Aleši Smrčkovi, Ph.D. a Ing. Janu Fiedorovi za jejich pomoc.

Literatura

[1] Paul Menage, P Jackson, and C Lameter. Cgroups. http://www.mjmwired.net/kernel/Documentation/cgroups.txt, 2008.

[2] Jan Fiedor and Tomáš Vojnar. Anaconda: A framework for analysing multi-threaded c/c++ programs on the binary level. http://www.fit.vutbr.cz/~vojnar/Publications/rv-12.pdf, 2012.

[3] Michael J. Eager. Introduction to the dwarf debugging format. http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf, 2012.

[4] Joseph Koshy. libelf by example. ftp://ftp2.uk.freebsd.org/sites/downloads.sourceforge.net/e/el/elftoolchain/Documentation/libelf-by-example/20120308/libelf-by-example.pdf, 2010.

[5] Tevi Devor. Pin: Intel's dynamic binary instrumentation engine. https://software.intel.com/sites/default/files/managed/62/f4/cgo2013.pdf, 2013.

[6] Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. Atomrace: Data race and atomicity violation detector and healer. http://www.fit.vutbr.cz/~vojnar/Publications/lvk-padtad-08.pdf, 2008.

[7] Stefan Savage, Michael Burrows, Greg Nelson, and Patrick Sobalvarro. Eraser: A dynamic data race detector for multithreaded programs. http://homes.cs.washington.edu/~tom/pubs/eraser.pdf, 1997.

[8] Klaus Havelund. Using runtime analysis to guide model checking of java programs. http://ti.arc.nasa.gov/m/pub-archive/archive/0177.pdf, 2000.